HLS for Face Classifier

Group Number: 11

Name	Roll Number
Aasneh Prasad	210101001
Sahil Mehul Danayak	210101092
Sanjana Siri Kolisetty	210101093
Dhanesh V	210101117
Ketan Singh	210101118

1 Description of the model

The model involves implementing **face detection** and **recognition** on a dataset featuring prominent Indian personalities, including Deepika Padukone, Ayushmann Khurrana, Kriti Sanon, Sushant Singh Rajput, Narendra Modi, and Carry Minati. The dataset comprises of **60** images for training and **18** for validation. Leveraging the `dlib` CNN face detector, faces were identified within the images and sorted into separate folders according to individual identities. Subsequently, embeddings were extracted from the images to prepare the training and testing datasets. These embeddings were fed into a simple `softmax` regressor with three layers, each optimized for predictive accuracy. Upon predictions, bounding boxes were applied around detected faces, with **annotations** indicating the respective person's name.

Table 1: Model Description

Task of model	Number of layers	Type of Layers
Face Classification	38	6

Table 2: Layer Types & their count

Layer Type	Count
Convolution2D	16
ZeroPadding2D	13
MaxPooling2D	5
Dropout	2
Flatten	1
Activation	1

2 Adjustments implemented to ensure the synthesizability of keras2c files

Modifications that have been instituted to uphold the synthesizability of keras2c:

- The incorporation of precise header files (in *Vivado* source files) has been undertaken to facilitate all function invocations within face_classifier.c, thereby ensuring their seamless execution. These header files include:
  - K2C_activations.c
  - K2C_core_layers.c
  - K2C_helper_functions.c
  - K2C_include.h
  - K2C_normalization_layers.c
  - K2C_tensor_include.h
- The initial definition of the k2c_tensor structure included a float*array attribute, which presented a challenge due to the non-synthesizable nature of pointers in hardware description languages (HDLs). To address this limitation, we determined the maximum size requirement of this array. It was found that the *dense_kernel* and *dense_kernel_array* were the objects of the structure that required a maximum capacity of up to 262,200 elements. Consequently, the attribute was redefined as float array[262200] to ensure synthesizability.
- The integration of function pointers had been initially employed to facilitate the invocation of activation functions such as k2c_softmax_func and k2c_tanh_func. However, the usage of function pointers renders the code non-synthesizable. The function calls were manually inlined at the locations where these activation functions were invoked, ensuring the synthesizability of the codebase.
- Redeclaration of variables such as size_t i and size_t j in *for loops* in various loops of different/same function call lead to ambiguity in variable resolution, making it difficult for the synthesis tool to determine which variable declaration to use and resulted in non-synthesizability of the code. Instead of redeclaring variables everytime, such variables were declared globally. By declaring these variables globally, they could be accessed by all relevant parts of the code without introducing ambiguity in variable resolution.
- *Removal of memcopy Functions :* The original code contained many instances of the memcopy function, which was not synthesisable in the top module. Thus, we have removed such memcopy functions and simulated their effects with pipelined loops to initialise arrays.

3 Adjustments implemented to generate HLS4ML report

- Changed the clock period from 5 ns to 10 ns and board part to xcvu190-flgb2104-1-i to match the keras2C implementation.

- After doing the above change, we got an error due to excessive memory usage in `nnet_dense_latency.h`, hence we removed all **array partition** pragmas in that file. We were still getting the same memory error for our model, which we think is due to the huge weight size ($\sim 2.6L$), hence we removed all the **array partition** pragmas in `my_project.cpp` file also after which it ran successfully.

- we got a `config_schedule -enable_dsp_full_reg=false` warning due to which we commented that part in `buildprj.tcl`. Also, we increased the `config_array_partition` to 409666.

4 Challenges & Resolutions

- **Insufficient BRAM in FPGA board:**

  The keras-2-c generated code was overfitting the BRAM even in the largest FPGA board. This was because the struct of k2c tensor was defined having an array of size 2.6 lakhs. To mitigate this and reduce the BRAM, we created 2 tensors one having array of size 2.6 lakhs and other having an array of size in thousands. This was done to prevent wastage of memory of 2.6 lakhs everywhere k2c tensor was defined and the entire array was not required.

- **Problems with C/RTL Cosimulation**

  1. Initially, the C/RTL cosimulation encountered a failure due to a memory error. The resolution involved increasing the stack space size to **16 MB** for C Simulation using the linker flag `-Wl,--stack,16777216`. This adjustment was made in the project settings at `project -> project settings -> simulation -> synthesis -> linker`. Additionally, all tensors and arrays were declared globally, and the `static` and `const` keywords were utilized for read-only arrays within the test suite file. These measures successfully resolved the memory error.

  2. Subsequently, the occurrence of the error message "`The OPMODE 0110X0X with CARRYINSEL 000 to DSP48E1 instance is invalid.`" was addressed by following the steps outlined in the documentation provided by Xilinx Support. Following the recommended actions, the C/RTL cosimulation executed without encountering further issues.

- **Huge time taken in C/RTL Cosimulation**
  One of the issues we faced was the significant time taken for C/RTL cosimulation. It would take around 1.5 hours for a single test case, which made it difficult to apply pragmas and check their effectiveness. To address this issue, we estimated the minimum and maximum iteration bounds by analyzing the input and used the `#pragma HLS loop_tripcount` for all the `for` loops. This allowed us to remove the "?" in the latency part of the synthesis report and obtain **bounds** on latency.

5 Optimizations done

1. `face_classifier_c.c`: In our codebase, we have employed **HLS unroll** pragmas to enhance parallelism in loops characterized by small loop bodies. For some loops,

we have used an **unroll factor** to strike a balance between latency and resource utilization. Additionally, we have utilized **HLS pipeline** directives for loops featuring longer loop bodies, with the objective of attaining an initiation interval of 1 for smaller loops.

Code Restructuring: Rather than defining arrays separately and subsequently performing memory copying operations into tensors, we have opted to initialize the arrays directly within the tensors. This leads to reductions in number of LUTS and BRAMS (one less array defined) and reduction in number of ALUs (by decreasing assignment operators). Further this also decreases the latency of the code.

Failed Approaches: We tried optimization using pragma HLS Interface, however, the data was in multiple cores of the CPU and thus the array could not be used as a memory interface with a certain depth. We also tried using pragma HLS Dataflow, however it gave an error as certain k2c_tensors were globalized.

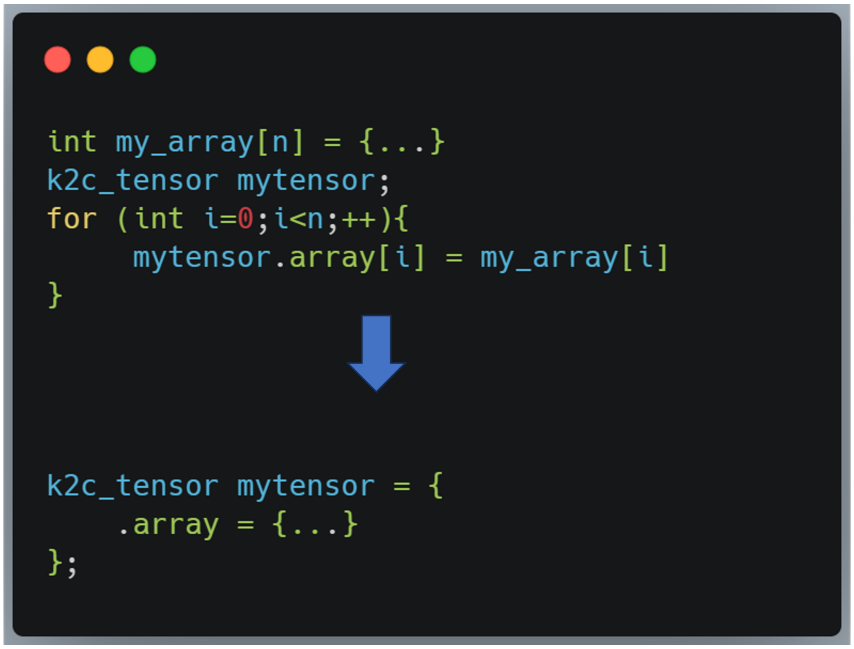

Figure 1: Array Restructuring.

2. `k2c_dense` in `k2c_core_layers.c`:

   For the starting optimisation, we have inlined the function `k2c_affine_matmul` in `k2c_helper_functions.c` for better stack space and to reduce function calls. This resulted in a much better-restructed code.

   In our codebase, we have employed `HLS pipeline` pragmas in the innermost loop to obtain the best possible Initiation interval. As was discussed in class, We have a three-level loop in our code base, which is a matrix multiplication. However, we were not able to take advantage of the reordering of loops from i-j-k to k-i-j as in our codebase because we have an array initialisation of output array in the upper two loops i-j, the exact code line is : `output->array[outrowidx+j] = bias->array[j];`

To accommodate this in the k-i-j loop format, we have to run another loop of i-j for the initialisation of the output array. The produced code runs in $O(N^3 + N^2)$, which has an optimised $O(N^3)$ loop in terms of memory, but it performs worse than unoptimised $O(N^3)$ due to overhead $O(N^2)$. Thus, we have avoided this optimisation.

The next optimisation we tried included `HLS Array_partition` pragmas to allow better data access. In doing so, we ran into an error. We tried to use block partition with factor 4 but could not run this pragma. The program ran into an unexpected error, and logs showed to inspect .dmp file. Using tool **WinDbg** we were able to inspect.dmp file. The analysis suggested that maybe the error was due to the mapping of different arrays to the core of the actual PC, but we were not able to come to any substantial conclusion.

3. `k2c_batch_norm` in `k2c_include.h`:

   - `pragma HLS pipeline` was used along with along with estimated trip count because of varied number of elements due to which it can't be unrolled.
   - Since the `offset` was always 1, we removed some of the *dead code* and commented it.
   - We chained all the operations together in a single line so that it can be executed efficiently.
   - `pragma HLS array_partition` was used with `block_factor=10` to allow parallel access to operations in `output[i]`,`stdev`, `gamma` arrays.

4. `k2c_dot` in `k2c_helper_functions.c`:

   - **Removing Macros**: Instances of `K2C_MAX_NDIM` in arrays `permA[K2C_MAX_NDIM]`, `permB[K2C_MAX_NDIM]`,`freeA[K2C_MAX_NDIM]`, `freeB[K2C_MAX_NDIM]` have been replaced with 5.
   - **Dead Code Elimination & Restructuring**: Many loops have been restructured to remove the inner loop and some conditional block. The inner loop runs for a fixed number of iteration, (estimated via `#pragma HLS loop_tripcount`), hence such modification was possible.
   - `pragma HLS loop_tripcount`: This helped in estimating the number of iterations of each loop. With the help of this, we identified loops of very small size such, which could have either been handwritten or optimized using `#pragma HLS loop_unroll`.
   - `pragma HLS PIPELINE`: Multiple pipelines have been incorporated in various loops to optimize the latency, as well as at places where the number of iterations was variable and hence unrolling was infeasible.
   - **Matrix Multiplication**: We have applied `pragma HLS PIPELINE` to the inner most loop of matrix multiplication.
     To optimize it further, an attempt to decrease the number of accesses to the arrays, we tried reordering the loops from `i-j-k` to `k-i-j`. However to achieve such reordering, we had to intialize the result array `A->array`, with zeros, however to do so we would need an overhead of $O(N^2)$, which performs worse than the unoptimized version.

5. `k2c_bias_add` in `k2c_helper_functions.c`:

   - `pragma HLS pipeline` was used along with along with estimated trip count because of varied number of elements due to which it can't be unrolled.
   - `pragma HLS array_partition` was used with `cyclic_factor=10` to allow parallel access to operations in `A->array` arrays but it increased the estimated time by a huge factor and it became more than the clock time. Thus it was dropped.
   - `pragma HLS DATAFLOW` could've been used but since the `k2c_tensors` were globalized, it gave an error as dataflow can only be applied for local variables.

6. `k2c_idx2sub` in `k2c_helper_functions.c`:

   - `pragma HLS pipeline` was used along with along with estimated trip count because of varied number of elements due to which it can't be unrolled.

7. `k2c_sub2idx` in `k2c_helper_functions.c`:

   - `pragma HLS pipeline` was used along with along with estimated trip count because of varied number of elements due to which it can't be unrolled.

6 Results

- **Latency and area overhead table for Baseline (Unoptimized)**

Table 3: Baseline (Unoptimized)

Design	LUT	FF	DSP	BRAM	latency(min/max)	Clock period
face_classifier_c.c	40437	41583	152	13175	3173423	10

- **Latency and area overhead table for Optimized**

Table 4: (Optimized)

Design	LUT	FF	DSP	BRAM	latency(min/max)	Clock period
face_classifier_c.c	73683	79305	202	1584	1849764	10

- `HLS4ML` **generated Latency and area overhead table**

Table 5: `HLS4ML`

Design	LUT	FF	DSP	BRAM	latency(min/max)	Clock period
face_classifier_c.c	19633	2268	9	303	1067261	10

- **Problems with** `HLS4ML` **generated report**
  - In `HLS4ML`,the firmware files were in C++ while we had to optimise our code in C language. This allowed `HLS4ML` to use `ap_type` for float datatype, which can't be used in C.

– In `HLS4ML`, they used `hls::stream` library due to which they were able to use `pragma HLS stream` and dataflow pragmas to reduce the latency and Initiation Interval, which are not available in C language (See Page 229 in Xilinx Vivado HLS manual.